Object Module Format Appendix D

Description of Object Module Format Version 1.3.0

D.1 INTRODUCTION

Moderate to large software programs are almost always compiled and/or assembled in more than one compilation unit (*module*), so that the entire program does not have to be compiled in one run. This is necessary to prevent compilation times and the memory required by the compiler from becoming unmanageably large. The result of compiling one source module is an *object module*, which contains generated code and initialized data along with information allowing it to be combined (linked) with other modules to make the complete program. A program called a *linker* combines the modules.

To allow a useful program to be composed of several separately compiled modules there must be a way to refer to global objects across module boundaries. At the source code level, global objects are referred to by textual symbols which are deemed by the compiler to be of global (inter-module) scope. These are called *global symbols*. Since the compiler/assembler has no knowledge of the address of a global object defined in another module, the symbol itself is embedded in the code or data.

The linker's main tasks are *resolution* and *relocation*. Resolution is the process of matching global symbols across modules, making sure all references to such a symbol refer to a single unique object, and replacing the symbolic reference with the address of the object. This involves filling fields in the object code with the addresses of global objects. Relocation is the process of stacking the code and data so they do not overlap in the address space. Each module may contain code or data for several address space *sections* (or *segments*), and each of these is treated separately (figure D-1). Thus each section from each module is relocated in the address space. The contributions to any given section from each object module are usually stacked contiguously. Relocation necessitates replacing address references in the code or data with the new address references to the relocated objects to which they refer[1].

Thus an object module must contain information about global symbols and all references to them, and information identifying all references to relocatable objects (whether identified by a symbol or not).

Most UNIX-based software development environments support the 'a.out' format for object modules [1], each environment and language adding its own extensions. This ancient format only supports fixed word widths and a single text and a single data section[2] (the latter divided into initialized and uninitialized), and has been heavily adorned with special features and extensions to allow it to support a plethora of specialized word widths and addressing modes for specific target machines. Because it is defined using machine dependent data types (like integer), it is not portable across host machines. Another more recent format, COFF [2], is cleaner and could support multiple named sections, but is still locked in to certain discrete word widths and fixed fields for relocatable addresses, and is not portable across host machines.

---

1. This paragraph conveys a somewhat simplistic view of references for th sak of clarity in the introduction. In fact, a reference is not always a legitimate address, but may be a bas from which an address is computed at run-time.
2. Called *segments* in a.out terminology.

Object Module Format

Inspired and motivated by the LIFE project [3] which supports a family of processors of arbitrary wordsize and with instructions having arbitrarily wide fields, a new object module format has been developed that represents code and initialized data as bitstrings, and can manipulate arbitrarily scattered bitfields within those bitstrings for the purposes of address relocation and symbol resolution. While providing the generality of code and data structure that LIFE requires, it does not add any special features specific to LIFE. In fact it is so generic that it could be used with any currently existing general purpose architecture,[1] and can provide a clean and general base for extension to support special features that may be required by other architectures. It also avoids artificial restrictions on quantities like the length of symbol names by providing variable length representations of all entities that are not, by nature, fixed in length.

Having been in active use in the LIFE project for over two years, the format has undergone a number of evolutionary improvements and is now fairly stable, except for debugging information which has only recently been defined in detail and may undergo more changes as the LIFE development tools are modified to support it. It is described in a machine independent manner in the ANSI C language by a set of header (.h) files and is supported by a set of machine-independent C macros.

The linking process has been briefly described only as background information to show the purpose and requirements of the object module, and will not be discussed further in this report.

Before delving into the object module format in detail, it is necessary to introduce some more concepts.

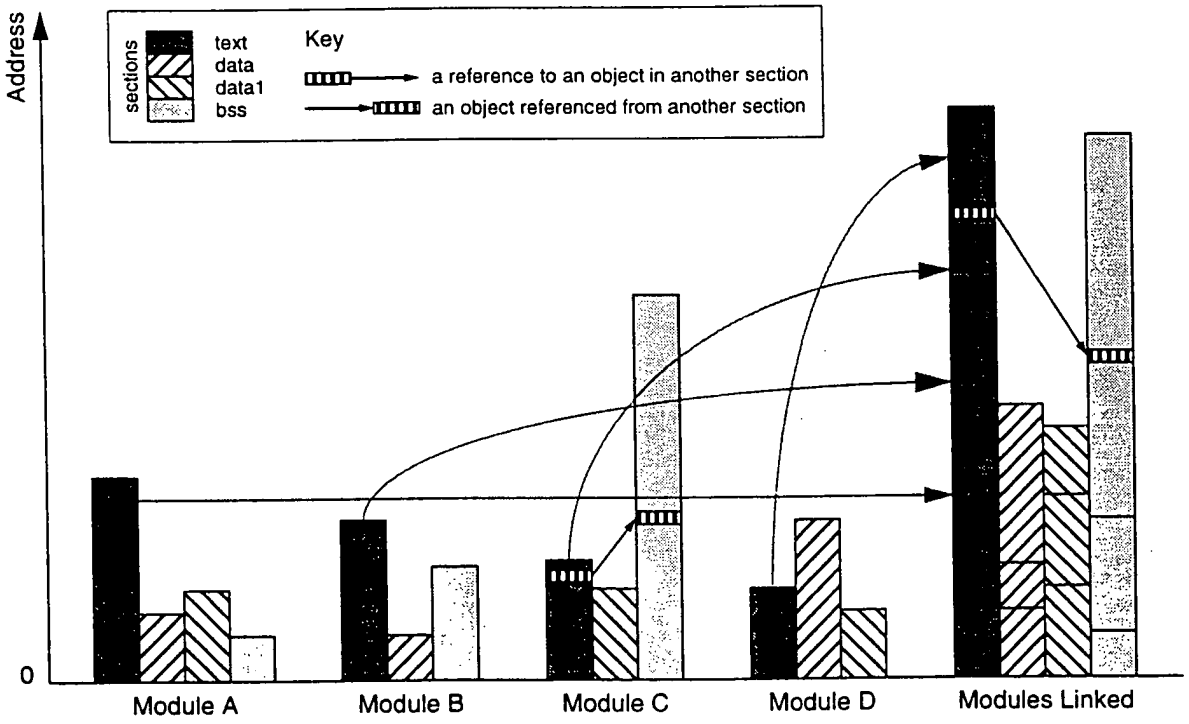

Figure D-1Relocation in Linking.

1. provided, of course, the compiler/assembler can generate object modules in this format.

D.1.1 Symbol Types

Two global symbol types have already been alluded to: definitions and references. There is also a third type that is required to support FORTRAN and is also used by many C compilers, the *common* symbol representing a *common block*. Each of these will be discussed in turn:

definition
A global symbol definition is a symbol with global scope that has a value associated with it. The value is usually an address in a specified section. The address is relocated according to the section it is in, and all references to this symbol are resolved to this address. Only one definition is legal for each global symbol in a program.

external reference
An external reference is a reference to a global symbol that is not defined in the module containing the reference. It is associated with a field in the code or data of one of the sections of the module. The reference may include an offset to be applied to the symbol value when it is resolved. A module may contain many references to the same symbol, each with its own offset. To simplify the symbol resolution part of linking, an entry is included in the global symbol table to represent all references in the module to that symbol. This symbol is called an *external* symbol.

common
A common symbol represents a global common block, which is a region of memory that may be referenced in several places in the program, but whose location is not yet defined anywhere (though a minimum size and a section may be specified in many places). It directly corresponds to the FORTRAN common block, and is used in C for *tentative definitions* (uninitialized data structures that may be defined in several places). When a common symbol matches a common symbol from another module, the two are merged into one common symbol having the larger size. The memory for a common is not allocated until the final stage of linking, after all modules have been loaded and the largest size for the common has been determined. Each common symbol is represented in the global symbol table by a single entry specifying the size of the largest occurrence in that module.

D.1.2 Various Forms of Object Modules

A program can be linked in many steps, according to the needs of the developer. All the object modules comprising the program may be linked at once to form the entire program, or some of them may be linked into a larger object module, and this module later linked with more modules, ad infinitum, until all original modules have been incorporated. To allow a linked module to be further linked, the resulting module must retain all the global symbolic and relocation information. A module that can be further linked is called a *relocatable object module*.

Once all the modules comprising a program have been linked together, an *executable* can be produced. An executable has the properties that all global symbol references have been resolved to address references, all commons have been defined (assigned an address) and an address in the code at which to begin execution (the *start address*) has been defined. The start address must have been specified in exactly one of the contributing object modules, or in the linker command line. Since an executable does not require further linking, global symbol information can be optionally discarded (but is often retained to help with debugging). Relocation information is still needed if a loader is to relocate the program to fit into the memory map of the machine that will execute it. An executable without global symbol information can be called a *relocatable load module* because it can be relocated but not further linked. However if the target machine has a fixed memory map and the program is to be loaded into a predetermined place (eg. if it is to be stored in ROM), the linker can locate all the sections in the executable at their final places and discard the relocation information. An executable without relocation information is called an *absolute load module* because all its address references are absolute.

D.2 OVERVIEW OF OBJECT MODULE STRUCTURE

One of the goals in designing the object module format was to avoid building in hard size limits on symbol names, tables and lists, except for unavoidable limits imposed by the number of bits used to index tables (these limits are set very high so they should never be a problem). So the object module uses a lot of variable length data structures to support very large object modules while avoiding the overhead of large fixed length tables in small modules.

Another goal is to permit fast linking in a minimum number of passes by a linker designed to run with limited memory in which it is not feasible to load all the object modules at once[1]. Thus the order of tables in the file corresponds to the order in which they are needed by a linker, and the size of any variable length table is available before any of the table data.

Yet another goal of the format is portability among host machines. This means an object module should be able to be copied to a different host machine and still be read by that host's linker or loader without any problems. Thus the format defines byte order in all complex structures (even multi-byte integers) and bit order in bitstrings and bitfield references. It is the responsibility of the software tools that read and write object modules to ensure they are correct, even if they have to swap byte order or twiddle bits. To this end a family of *logical data types* has been defined along with a suite of machine independent macros to read and write their instances between file buffers (byte arrays mirroring the data in the files) and machine dependent internal representations.

As discussed in the introduction, the object module format supports arbitrary length bitfields in the code and data, which are represented by bitstrings. It also supports full 64 bit addressing.

Figure D-2 shows an overview of the structure of the object module. The object module is divided into four *partitions*, all but the first being of variable length. The first partition, the *header*, is of fixed length (128 bytes), and contains information on the sizes and offsets of the other partitions, as well as various data pertaining to the entire module. The second partition, the *global partition*, contains all the information needed to resolve symbol references and match and relocate sections by name, and also indexing information for the other two partitions. It consists of the following tables: *string table*, *section table*, *global symbol table*, *linking history*, *source table*, and *scatter table*. These are all of variable length and are indexed by information in the header. The third partition, the *binary partition*, contains the actual bitstrings representing the code and initialized data, along with a *reference table* (list of references) describing bitfields within each of these bitstrings. The binary partition is organized per section, and thus is indexed by the section table in the global partition. The fourth and last partition, the *source file access partition* (often abbreviated as the *source partition*), contains information related to debugging and is entirely optional (a module without debug information has no source file access partition). It is organized per *source file*[2] and is indexed by the source table in the global partition.

The next four chapters describe each partition in some detail, giving reasons for certain design decisions. An appendix contains full details defining the logical types and the object module format, in the form of the actual C header files.

---

1. This being the case, it should be pointed out that in fact the existing LIFE linker does not make use of this capability, rather it assumes there is plenty of virtual memory and *does* load all modules at once in the interest of speed.
2. This might more correctly be referred to as a *compilation unit*, since a compilation usually involves several included sourc files as well as the basic source file. It is really the composite sourc string that is parsed by the compiler or assembler that generates a first-gen ration (not linked) object module.

Object Module Format

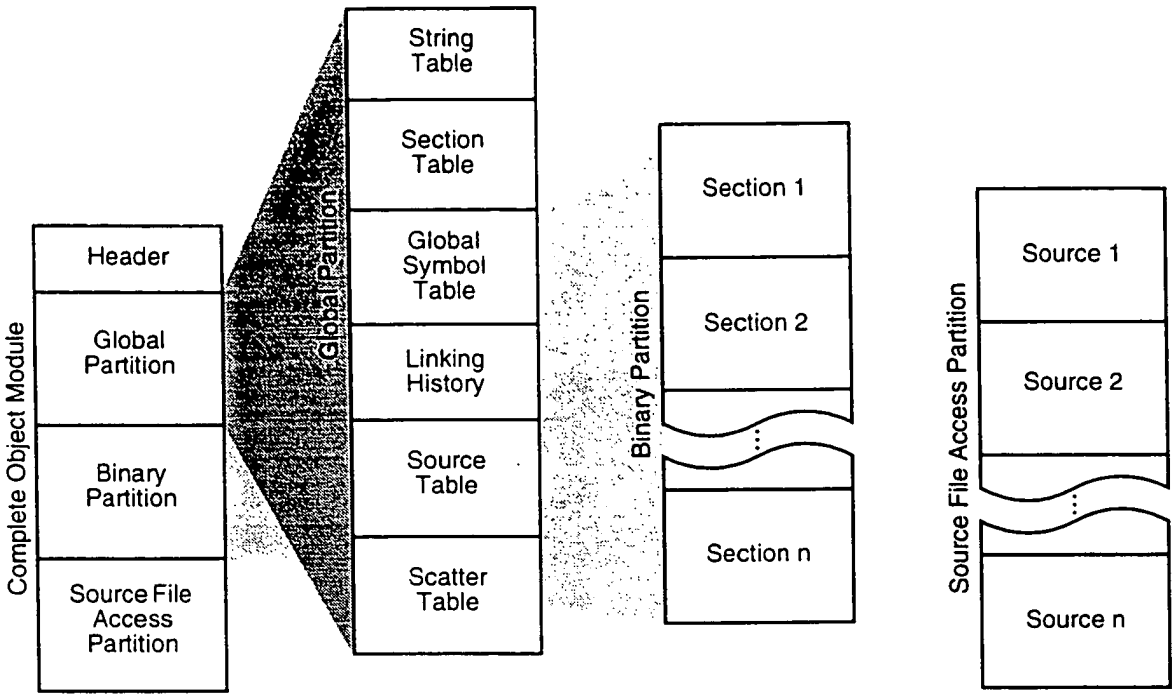

Figure D-2Structure of the Object Module.

D.3 HEADER

The module header is a fixed size (128 byte) partition at the beginning of the file with fixed length fields in fixed locations. A linker or loader can load this in one disk access, and interpret it to obtain the information on the sizes and offsets of the remaining partitions within the file. The header also contains some information pertaining to the module as a whole.

D.3.1 Magic String

The magic string is an eight character fixed-length string that identifies the file as a LIFE object module. It permits various programs such as a linker, loader, or the UNIX 'file' program, to recognize the file type. The value of the string is 'LIFE_Obj' and it is not NULL terminated. In order for the UNIX 'file' utility to recognize this, a line like the following must be added to the /etc/magic file on the system:

```
0       string          LIFE_Obj        LIFE Object Module
```

D.3.2 Object Module Format Version

The object module format version is identified in two ways, one machine checkable and one for the benefit of humans using an object module dump program[1]:

1. A program 'LifeDump' exists to parse and print the contents of an object module in human readable form.

Version Number
A 2 byte decimal version number of the form dd.d.d identifying the version of the object module format. This always follows directly after the 8 byte magic string so it can be located easily in an object module of any version. The first byte is treated as a decimal number up to 255. The second byte is treated as 2 decimal digits. The current version, described in this report, is '1.3(.0)' (the trailing '.0' can be omitted).

Version String
An offset into the string table in the global partition representing an arbitrary NULL terminated string that can be printed to identify the object module version to a human reader. It should contain, but is not limited to, the version number that is stored in the decimal version number field. The string currently stored here is 'LIFE Object file version 1.3 of 25 Jul 1994'. Note that the version string cannot be read until the global partition has been read, hence the retention of the decimal version number stored directly in the header.

D.3.3 Target Machine Identification

One of the problems with a generic object module format that is used for many different target machines is ensuring that all the object modules being linked to build a program were compiled/assembled for the same target machine, and that a program to be loaded and run on a machine was indeed compiled/assembled for that machine. This is done is two ways, one machine checkable and one for the benefit of humans using an object module dump program:

Machine Checksum
A 32 bit integer inserted by the compiler or assembler that generates an original object module. It is intended to be some kind of checksum computed from information identifying the machine type, but its semantics are not specified by the object module format. In the case of LIFE it is computed from the machine description (.life) file. A linker should check that all input modules have the same checksum, and if not flag a fatal error. If they are all the same it should proceed with linking and insert the same checksum into the module it produces. Note that a linker need not understand any semantics associated with this checksum. A loader should check that the checksum is the same as one computed for the machine it is loading to. The loader and the compiler or assembler must agree on the semantics of the checksum.

Machine Identification String
An offset into the string table in the global partition, representing an arbitrary NUL terminated string that can be printed to identify the target machine to a human reader. For example, it might be the name of a machine, like SPARC. In the case of LIFE it is the basename of the machine description file, eg. 'life25' for a module compiled and assembled with machine description file 'life25.life'.

D.3.4 Flags

The header contains a 16 bit integer field containing several binary (bit) flags indicating properties of the object module. The following flags are defined:

is_exec
Indicates that the object module is an executable program. To be executable the following three conditions must hold:

- There must be no unresolved symbol references.
- There must be no common symbols (ie. all commons must have had memory allocated).
- The module must contain a start address.

has_start
Indicates that a start address is defined in this module (see section 4.3.5). This is the address at which the program is to begin execution.

is_abs
Indicates that the module is an absolute load module. It contains no relocation information so all addresses are absolute, and the base address of each section is specified in the section table. Such a module is suitable for programming a ROM.

is_compressed
Indicates that the text section of the object module is in compressed format. This is a LIFE specific flag, though a flag of similar nature may be needed in other architectures as well.

D.3.5 Start Address

The start address is the address at which the program is to begin execution. It is usually relocatable with respect to a code section. Thus there are two parts to the start address: the section, and the address relative to the base of that section. If no section is specified the address is absolute (not relocatable).

D.3.6 Partition Offsets

The header contains information for finding the other partitions in the file. This consists of the offset from the beginning of the file to the start of each partition, and the size (in bytes) of each partition. The global partition offset is not included because it is constant (128), beginning right after the fixed size header.

D.3.7 Global Partition Index

The header also contains information for extracting the various tables from the global partition. It contains the offset to each part from the base of the global partition (not the base of the file) allowing for the global partition to be loaded into a dynamically allocated buffer once its size is extracted from the header. The parts of the global partition are: *string table*, *section table*, *global symbol table*, *linking history*, *source table*, and *scatter table*.

Each of these tables contains a number of fixed size records (in some cases the records are simply bytes, but often are structures), so the size of each part is given as the number of records rather than bytes. This makes extraction easier and permits easy determination of table sizes for creation of internal data structures composed of records. The sizes of the tables are given as follows: string table (bytes), section table (number of sections), global symbol table (number of symbols), linking history (bytes), source table (number of compilation units), scatter table (bytes).

D.4 GLOBAL PARTITION

The global partition is a variable length partition containing all the information necessary to match and relocate sections by name and resolve symbol references. It also contains indexing information for the binary and source partitions.

D.4.1 String Table

An object module contains many strings, mostly names of symbols. In order to avoid imposing a hard limit on the length of symbol names and other strings, all strings that are part of the object module are stored in a variable

length *string table*, which is a contiguous string of NULL (\0) terminated strings, each unique string appearing exactly once. Strings are then identified in the rest of the object module by simple integer indices which are offsets from the base of the string table (figure D-3). These indices are called *string identifiers*. For convenience of implementation in software tools that generate object modules, there is always a null string at the base of the table, so a zero string ID represents a null string by definition. These tools use a hash table to ensure uniqueness of strings in the table.

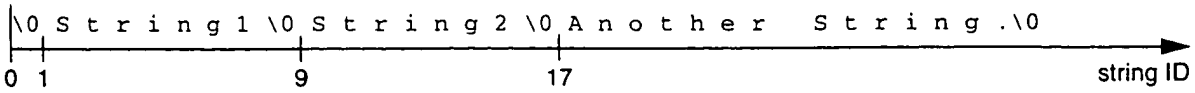

Figure D-3Structure of the String Table.

D.4.2 Section Table

The section table is an array containing an entry (a *section descriptor*) for each address space section[1]. Within the object module a section is identified by a *section identifier* (or *section ID*) which is the index of the section descriptor in the section table, starting at 1 (a zero section ID indicates that no section is specified).

The section descriptor contains a name (identifying the section across modules), relocation information, and indexing information for the section's entry in the binary partition. In LIFE, modules have four sections which are the "standard" ones generated by the LIFE C compiler, namely 'text', 'data', 'data1' and 'bss'. But the object module format treats sections generically. It does not associate any semantics with section names. Additional named sections can be added, up to a maximum number of 255.

The fields of the section descriptor are described in the following subsections:

NAME

The name is a string identifier (offset in the string table) identifying a section for the purposes of matching sections during linking or loading them into a computer's memory or a simulator. Each section in a given module must have a unique name, and this name is global in scope across the modules in a program. A linker will merge together all the sections with the same name from all of its input modules. Neither the object module nor the linker make any distinction between instruction and data address spaces. Every section is treated as a separate address space, and it is up to the loader or simulator, prior to running the program, to locate each section in the proper address space.

BASE ADDRESS

The section base address is the address at which the section begins in its address space. Usually zero (because each section is treated as if it has its own address space), it can be non-zero, for example, in a module that has already been relocated for loading into a fixed address.

SIZE (in Address Space)

The section size is the size of the section in terms of the number of addresses it spans.

---

1. Alternatively called a *segment*. These terms may be used interchangeably.

CACHE BLOCK SIZE

If a module is intended for a target machine with a cache, it is usually desirable for objects to be aligned on cache block boundaries to minimize the number of cache blocks they span. Many architectures also require that objects be aligned to boundaries determined by the largest chunk of memory that can be accessed in a single machine instruction (sometimes called the *wordsize*). This latter requirement is independent of whether the machine has a cache, but it is necessary that the cache block size (if there is a cache) be a multiple of the wordsize[1]. Proper alignment of objects to wordsize and cache block boundaries is ensured within a module by the compiler or assembler that produces it. But when modules are relocated (during linking or loading) the proper alignment may be lost unless the linker or loader takes alignment into account to ensure that each section still begins on a cache block boundary[2]. It is often necessary for a linker to leave a gap in the address space at the end of a section to ensure the contribution from the corresponding section in the next module is properly aligned. To this end, the cache block size is included in the object module on a per-section basis.[3] A linker must check that sections it is concatenating have the same cache block size, and a loader must check that the cache block size of the memory a section is to be loaded into is the same as that specified for the section in the object module. The cache block size is specified in terms of address space, ie. the number of distinct addresses that fall into one cache block. If the target machine has no cache, the word size should be specified in place of the cache block size. The minimum legal value is one. A cache block size of zero is illegal.

MEMORY WIDTH

The memory width of the section is the number of bits addressed by a single memory address. For data memory this is usually eight. For instruction memory it depends on the architecture (in LIFE this can get up to several hundred quite easily). This information is only required in the object module so that a linker can determine how many padding bits to insert into the bitstring to cover a gap in addressing created by alignment.

BINARY PARTITION INDEX

Two parameters indicate the location of the bitstring containing the code or initialized data for the section (if present). The bitstring offset is the offset from the base of the binary partition to the start of the bitstring. The bitstring length is the number of bits in the bitstring, from which the number of bytes can also be determined. For the text section of a compressed object module, the bitstring length may not be the same as the product of the size and memory width. This is because, the bitstring length indicates the length of the bitstring that excludes the padding bits that might have been added to make the bitstring length a multiple of the instruction cache word size. This is necessary to make ls+ (the machine level simulator) capable of decoding the instructions statically.

Two more parameters indicate the location of the reference table corresponding to the bitstring (this is a list of references to bitfields in the bitstring, and will be described in section 5.2). The reference table offset is the offset from the base of the binary partition to the start of the reference table. The reference table size is the number of references (entries) in the table.

D.4.3 Global Symbol Table

The global symbol table is a list, in random order, of *symbol descriptors* defining all the symbols of global scope

---

1. Responsibility to ensure this rests with the machine architect.
2. Since the cache block size is always a multiple of the wordsize, ensuring alignment on a cache block boundary also guarant es alignment to the wordsize. A linker therefor need not explicitly tak into account the wordsiz , so only the cache block size is specified in th object modul .
3. In reality it would be p r-address-space but, for r asons discussed earlier, each section is treated as a separate address space)

(global symbols) in this module, along with information about each one. Each symbol has a unique name. When modules are linked together symbols of the same name from different modules are merged together (resolved) into one symbol. The fields of the symbol descriptor are described in the following subsections:

NAME

The name of a global symbol uniquely identifies that symbol throughout the entire program. It is a string identifier referring to a string of arbitrary length that actually resides in the string table.

RELOCATION TYPE

The relocation type is a set of flags indicating what type of symbol entry this is and hence what should be done with it. It actually pertains more to resolution than relocation (but the name is historic and has stuck). References in the reference tables of the binary partition also have a relocation type which is identical in form (see section 4.5.2 on page 13). The flags in the relocation type are:

is_extern
Indicates that this module contains one or more references to a global symbol of this name which must be defined externally. If the *is_common* flag is not also set, then there are not even any tentative definitions for this symbol, and the *value* and *section* fields of the symbol descriptor are meaningless and will usually contain zero.
If the *is_extern* flag is not set, the entry is a global symbol definition, in which case the *section* field indicates the section it is defined in and the *value* field contains the symbol's value (an address relative to the base of the section).

is_common
Indicates that this entry defines a common block, or tentative definition. It is always used with the *is_extern* flag. When this flag is set the *value* field of the symbol descriptor contains the size (in address space) of the memory space needed by the object this symbol represents, and the *section* field is a suggestion as to the section it should be defined in (if zero, no section is suggested and the linker must choose).

is_signed
Indicates that the *value* field of the symbol descriptor contains a 64 bit signed value. Not presently used in the symbol descriptor (but used in reference tables).

is_local
Indicates that the symbol is local (in scope) to this module. Symbols in the global symbol table are never local so this flag is never set. It is used in reference tables (which contain references to symbols, among other things) and local symbol tables.

The relationship between these flags and the symbol types described in section 4.1.1 on page 3is summarized in table D-1 on page 15.

SECTION

A symbol definition must also specify the address space section the symbol is defined in, in order that the symbol can be properly relocated with respect to that section. If the symbol definition is absolute (not relocatable) the section ID is specified as zero. In the case of a common symbol (tentative definition), a non-zero section ID is just an optional suggestion to the linker, to be followed unless a definition for that symbol exists elsewhere in the program or another tentative definition (common) specifies a different section. In the case of an external

reference, the section ID is meaningless (and is ignored, though a linker may issue a warning if it is not zero).

ALIGNMENT

The alignment field specifies how the memory allocated for a common symbol should be aligned in the address space of the section it is allocated in. It must be located at an address that is a multiple of the alignment specifier. The alignment specifier must be an integral divisor of the cache block size of the section, otherwise the proper alignment may be lost when the section is relocated. A linker is required to issue a fatal error if this condition is not met. For a common symbol the alignment specifier must be non-zero, and a value of one indicates that no alignment is required.

For non-common symbols the alignment specifier is always zero.

VALUE

The meaning of the *value* field of the symbol descriptor depends on the symbol type, as follows:

definition
The value of the symbol. Usually a relocatable address in the section specified by the *section* field of the symbol descriptor, but an absolute number if the section ID is zero. This is a 64 bit integer. It is usually unsigned, but may be signed if the *is_signed* flag is set in the *relocation type* field of the symbol descriptor.

external reference
Meaningless (ignored) and usually zero.

common
The size of the memory required (in address space) by the largest tentative definition in the module. Always unsigned.

D.4.4 Linking History

The linking history of a module is a sequence of strings separated by NULL characters, much like the string table except that there is no NULL character at the beginning. Each string represents the time and version stamped linking command used in creating this module, most recent first. For example:

```
Wed Dec 16 12:10:04 PST 1992;LifeLink version 1.21 of Nov 24 1992;/home/
kiwi/ross/life/linker/dev/LifeLink -adjunct -exec -start=__start -
xref=csuite.symt -lib=lib/life25 -o=csuite.lx csuite.lo
```

The date, version, and command fields are separated textually by semicolons. This information can be read by a human using an object module dump program (see footnote 1 on page 5), and/or parsed by a debugger.

First generation object modules (produced by a compiler or assembler, not by a linker) have no linking history, thus this table is empty (has size zero). Each linker invocation inserts its own information at the beginning of the linking history of the module it is producing, then appends the information (if present) from each of its input modules in the order in which the modules were (or will be) loaded.

D.4.5 Source Table

The *source table* is an array containing an entry (a *source file information descriptor*, or just *source descriptor*)

for each source file[1]. Within the object module a source file is identified by a *source identifier* (or *source ID*) which is the index of the source descriptor in the source table. starting at 1 (a zero source ID indicates that no source file is specified). The source table may contain up to 65535 entries.

A first generation object module has only one entry in its source table. When object modules are linked their source tables are concatenated so that a record is preserved of each primary source file in the linked module, no matter how many generations of linking the program goes through.

The pathname in the source descriptor is a string ID referring to a string in the string table. The source descriptor also contains offsets into the source file access partition for the link map, local symbol table, and generic textual debug information; and sizes for each of these in terms of number of entries, as follows: link map (number of sections), local symbol table (number of symbols), generic textual debug information (bytes).

D.4.6 Scatter Table

The bitfields (for example the address fields in the bitstring of the text section) in the object module are not necessarily contiguous. Thus the linker will not be able to gather all the bits of the address field by just knowing the position and width of the field. It in fact needs the entire bit map. The *scatter table* provides such a map. Bitfields in a bitstring are specified by an offset in the bitstring and a scatter id which is an index into the scatter table. The scatter id indicates which scatter descriptor to use in getting the bits of the bitfield together. The scatter table has the following parts:

NUMBER OF ENTRIES

This will be the number of scatter descriptors in the scatter table.

SCATTER DESCRIPTORS TABLE

The number of entries field is followed by the appropriate number of scatter descriptors. Each *scatter descriptor* has the following format: an *unsigned integer* indicating the number of triples in the descriptor followed by a *sequence of triples*, each of the form *(dst_offset, width, src_offset)*. Here, *dst_offset* refers to the offset in the destination (where the address field is finally gathered) field, *src_offset* refers to the offset in the source field (it is the address (in the address space of the section) w.r.t. which the offsets in this table are specified.), and *width* is the number of bits that goes into the destination from the source at the given offset. The offsets have to be positive.

For example, let us say we have a scatter descriptor with three triples (0, 7, 3), (7, 4, 15), (11, 5, 23). Let us say the source field is at position 320 in the bitfield of the text section. To get the bits of the actual address field, we do the following: bits 0 through 6 of the address field are from positions 323 through 329 from the bitstring, bits 7 through 10 of the address field are from bits 335 through 338 of the bitstring, and bits 11 through 15 of the address field are from bits 343 through 347 of the bitstring. Thus the address field has length 16.

D.5 BINARY PARTITION

The binary partition is a variable length partition indexed by section identifier containing, for each section in the section table, a *bitstring* and a *reference table*. The offset and size of each bitstring and reference table is

1. This may be better termed a *compilation unit*. It is identified by the *primary source file* which is the one named in the compilation or assembly. The compilation unit may involve other source files that were textually included in the compilation or assembly by a directive in the primary source file. In C the primary source file would have the '.c' extension in its name, and included files would have th '.h' ext nsion.

stored in the section table in the global partition, discussed earlier. The bitstring contains the actual binary code or data for the section it represents, while the reference table identifies and describes *bitfields* within the bitstring. These bitfields contain objects in the code or data that need to be later identified, such as relocatable address references and references to externally defined symbols.

D.5.1 Bitstring

The object module treats data and instructions as bitstrings of arbitrary length. A bitstring is packed into a byte array with the most significant (MS) bit in the most significant position of the byte at the lowest address of the array, as illustrated in figure D-4 (the entire shaded region is the bitstring). If the bitstring fills the first byte, it continues from the MS position of the byte at the next higher address. After the least significant (LS) bit, any unused bits in the top byte of the array are ignored and can have any value. A bitstring can thus be designated by its base address (byte address) and length (in bits). The bitstring in figure D-4 has length 38.

A bitfield is identified by the base address of the bitstring and the bit offset and scatter descriptor of the bitfield. The scatter descriptor explained in section 4.4.6 has all the necessary information to gather the bits of the bitfield in the right order. It is important to note here that the bits of a bitfield need not be contiguous in the bitstring. Figure D-4 shows an example of bitfield whose bits are contiguous.

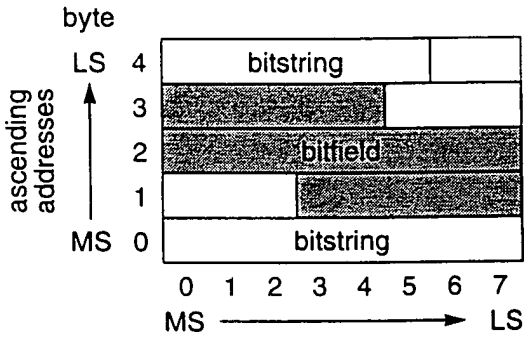

Figure D-4Bitstrings and Bitfields.

D.5.2 Reference Table

The *reference table* is a list of *reference descriptors* associated with a bitstring. Each reference descriptor refers to a bitfield in the bitstring. The list of reference descriptors is in the same order as the corresponding bitfields occur in the bitstring (based on the bit offset). This ordering allows sequential processing by a linker that doesn't have enough memory to load the entire bitstring and reference table at once. It can treat each as a serial stream instead of an array.

The following subsections describe the fields of the reference descriptor:

POSITION

The position is the bit offset of the base of the bitfield within the bitstring. The bitfield in figure D-4 has position 11.

SCATTER ID

This is an index into the scatter table where we can find the scatter descriptor that has information about the way the bits of the bitfield are scattered in the bitstring. The scatter descriptor corresponding to the bitfield in figure D-4 will have a single entry (0, 18, 0). Note that the src_offset here is 0. The actual offset in the bitstring is obtained by adding the position and the src_offset together, thus yielding 11 in this example.

RELOCATION TYPE

The relocation type is a set of flags indicating what type of reference this is and thus how it should be resolved and/or relocated. Its form is identical to the relocation type in the symbol descriptor (see page 10). The flags have the following meanings in the reference descriptor (slightly different from their meanings in the symbol descriptor):

is_extern
Indicates that this is a reference to an externally defined symbol, with a signed or unsigned offset. The symbol name (a string ID) is in the *symbol* field of the reference descriptor. The bitfield itself contains the offset to be applied to the symbol value when the symbol reference is resolved (signed or unsigned, depending on the *is_signed* flag).
If this flag is not set, the reference is already resolved (it may or may not be associated with a symbol), and the bitfield contains the actual value referred to. The value may still be relocatable. If so, the *section* field of the reference descriptor defines the section it is defined in (zero if not relocatable). The *symbol* field may or not contain a string ID. In a resolved reference the symbol is no longer needed for linking, but might be retained for use by a debugger.

is_common
Indicates that this is a reference to a common symbol (a tentative definition) with a signed or unsigned offset. It is really no different from an external symbol reference. This flag is only set in addition to the *is_extern* flag, and everything said above for the *is_extern* flag applies. This flag is actually redundant in the reference descriptor because the linker needs only to look at *is_extern* (commons are identified in the symbol table), but it is retained to keep the relocation type the same for symbols and references.

is_signed
Indicates that the content of the bitfield is a signed 2's complement integer, the most significant bit being the sign. This is usually true if the *is_extern* flag is set, but never true otherwise. One might suppose that this flag could be eliminated and offsets (indicated by is_extern) always be signed, but this might be too restrictive. Allowing the possibility of unsigned offsets permits the use of large offsets spanning more than half the address space representable by the number of bits in the bitfield (the bitfield width may be smaller than the width of an address) and helps keep the object module format generic.

is_local
Indicates that the symbol is local in scope to the source file (compilation unit) it is defined in. Only used for definitions, since local symbols are always defined. The symbol itself may be absent. Indicates to a debugger that, if a symbol is present, it is defined locally to the source file indicated by

the *source* field of the reference descriptor, and thus should be looked up in the local symbol table of that source file rather than the global symbol table. This flag is actually redundant because the *source* field of the reference descriptor indicates the same with a zero value if the symbol is global. It is included in the relocation type to permit a quick determination of the type of the reference by a bitwise test of combinations of flags.

Table D-1 summarizes how the relocation type flags other than *is_signed* are used in various situations. The *is_signed* flag is usually set only for external references (although it can legally be either set or clear in any of these situations).

Table D-1Relocation Type Flag Combinations in Symbols and References.

	Global Symbol Table	Local Symbol Table	Reference Table
global definition	(no flags)	N/A	(no flags)
local definition	N/A	is_local	is_local
common (tentative definition)	is_extern + is_common	is_extern + is_common	is_extern + is_common
external reference	is_extern	is_extern	is_extern

SECTION

If the bitfield contains an address, the *section* field indicates (by a non-zero section ID) the section it is defined in and must be relocated with respect to. If the section ID is zero, no relocation is required (in this case the bit-field contains either an absolute address or something other than an address). References to non-relocatable bit-fields are not necessary for linking, but might be useful for debugging, hence they are not prohibited.

SYMBOL

The *symbol* field contains a string identifier identifying a symbol associated with this reference. A reference need only contain a symbol if it is an unresolved symbolic reference, otherwise this field can contain zero (null string). A symbol is not needed for relocation, but may be retained for debugging purposes.

SOURCE FILE

The *source* field contains a source identifier identifying the source file (compilation unit) associated with a local reference. Such references have the *is_local* flag set in the *relocation type* field of the reference descriptor. For non-local references (in which *is_local* is not set) the source ID is zero. The source ID is used by a debugger when there is a symbol associated with the reference, to determine which symbol table to look it up in.

D.6 SOURCE FILE ACCESS PARTITION

The source file access partition (or source partition) is a variable length partition indexed by source identifier containing, for each source file (or compilation unit), a *link map*, a *local symbol table*, and *generic textual debug information*. The offset and size of each of these in the source partition is stored in the source table in the global partition as discussed earlier (section 4.4.5 on page 11).

D.6.1 Link Map

The link map shows the location and size in the address space of the contribution to each section from *this* source file. It is in effect a row of the overall link map, which may be pictured as a sparse array whose columns represent sections and whose rows represent source files. Using the overall link map, a debugger can determine the source file in which the code or data at a given address is defined. A fictitious link map is shown in table D-2. Each row represents the link map contribution from a given source file. Not all source files have the same sections, which is why it is a sparse array. In table D-2, shaded cells represent entries not included in the link map (eg. module_b.c has no data or data1 sections).

Table D-2Link Map.

Source File	text		data		data1		bss	
	base	size	base	size	base	size	base	size
module_a.c	0000	0084	0000	0010			0000	0048
module_b.c	0084	0012					0048	856E
module_c.c	0096	056F	0010	0A06	0000	00D8		
module_d.c	0605	0124	00A16	0047				
module_e.c	0729	00F5			00D8	0012		
module_f.c	081E	09E3	0A5D	0004	001F	000A	85B6	0103
module_g.c			0A61	0020				

The link map contribution from a source file is just a list of link map entries, which correspond to cells in table D-2. Each link map entry describes the base address and offset for one section of the current source file. The section is indicated by its section ID. The base and size are 64 bit addresses.

D.6.2 Local Symbol Table

The local symbol table is a list, in random order, of *symbol descriptors* describing all the symbols defined locally in this source file (excluding symbols of global scope, which are included in the global symbol table).

The symbol descriptor has the same data structure as in the global symbol table (see section 4.4.3 on page 9), but the meanings of the fields are a little different, so they are described briefly here.

NAME

The name of a local symbol identifies the symbol throughout the source file it is defined in.

Each local symbol has a name that is unique within the source file[1] (but not necessarily throughout the program). Another way to look at it is in terms of intersecting namespaces: each source file has its own symbol namespace which contains the global namespace. This is illustrated in figure D-5.

The name is a string identifier referring to a string of arbitrary length that actually resides in the string table.

1. That is, the compilation unit. High level languages such as C have several nam spaces within a module, but a compiler will modify names to map all symbols that end up in the object modul into on namespace.

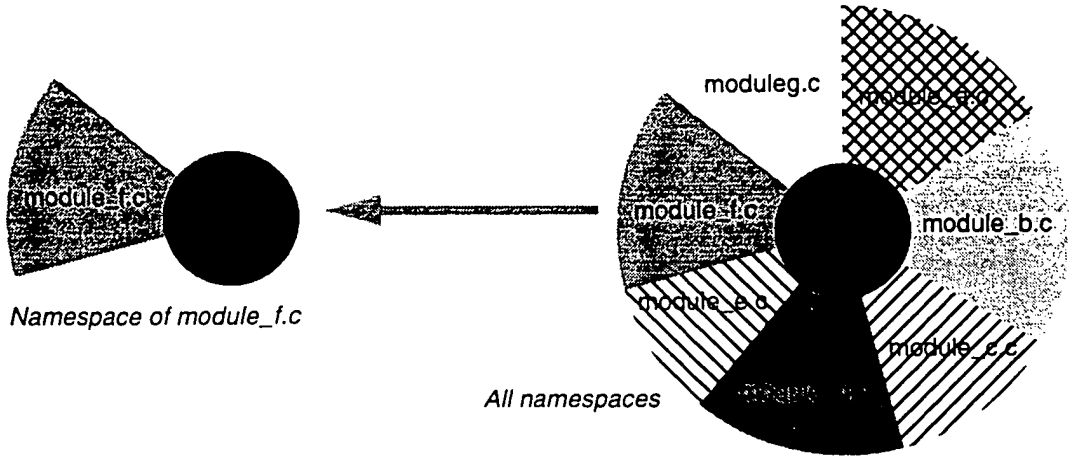

Figure D-5Overlapping Symbol Namespaces.

RELOCATION TYPE

The relocation type is not really needed in the local symbol table, but is retained for the convenience of having the same data structure for local and global symbols, and the flags are thus set appropriately. Because all local symbols in the object module are definitions, the *is_extern* and *is_common* flags are never set. The *is_signed* flag is not currently used because all symbol values, being fully defined, are inherently unsigned addresses or absolute quantities the linker doesn't need to perform any arithmetic on. The *is_local* flag is, appropriately, always set.

SECTION

The *section* field contains a section identifier specifying the address space section that the symbol is defined in and therefore must be relocated with respect to. If the symbol definition is absolute (not relocatable) the section ID is specified as zero.

ALIGNMENT

As alignment only concerns common symbols, which are always global, this field is not used in local symbol tables. It's value is always zero.

VALUE

The value of the symbol is usually a 64 bit relocatable address in the section specified by the *section* field of the local symbol descriptor, but it is an absolute number if the section ID is zero.

D.6.3 Generic Textual Debug Information

The generic textual debug information is a catch-all through which preceding levels in the compilation trajectory can pass information to succeeding levels. The linker does not need to interpret this in any way, but must retain the association with the compilation unit the information came from, hence it's location in the source partition.

The format of this information is intended to be ASCII text. But the object module format in fact imposes no restrictions on it. It is identified by it's offset and size (in bytes) in the source file access partition.

D.7 FUTURE CHANGES

Having been in active use in the LIFE project for over two years, this object module format has undergone a number of evolutionary improvements and is now fairly stable, except for debugging information which has only recently been defined in detail and may undergo more changes (as yet unforeseen) as the LIFE development tools are modified to support it. Nevertheless, a couple of minor changes to the relocation type flags may be introduced shortly, so they are briefly discussed here.

Currently, relative addressing modes are not supported in relocation, since all the work has been directed toward LIFE and LIFE does not use them. These modes require a linker to take into account the location of the reference as well as the location of the object being referenced. The object module format could easily be modified to support relative addressing with only the addition of a new *is_relative* flag in the relocation type, which would be used only in reference descriptors. This flag may soon be added to the relocation type.

Recent experience has cast into doubt the reliability of overflow checking (by the linker) in relocating and resolving references (as distinct from relocating sections), due to the fact that compilers sometimes do part of an address calculation and store a partial result as a reference, leaving the calculation of the actual address of the referenced object to be completed at run-time. Thus legitimate address references are sometimes outside of the address space (eg. less than zero) and require modulo arithmetic, and it is not clear whether relocation of any given reference will cause the final address actually referenced to be outside the address space. Thus the linker can only reliably check for overflow in section relocation (to ensure that sections remain fully within the address space), but not in relocation of address references. The *is_signed* flag in the relocation type is really only needed for overflow checking in resolved references, since without overflow checking all arithmetic on bitfields is modulo the bitfield width so the sign doesn't matter. Thus this flag may soon be rendered redundant and eventually removed from the relocation type.

D.8 CONCLUDING DISCUSSION

An object module format has been described which represents code and data as bitstrings, with variable length bitfields for relocatable entities or external references. Designed to be very generic, it supports multiple named sections and imposes no limits on the length of symbol names or the sizes of tables, other than those imposed by the width of integers used as indices (those limits are extremely generous). Addresses are represented in 64 bits, so present and upcoming 64 bit architectures can be supported. Also designed to be portable across development host machines, the format is described by header files in ANSI C, and specifies byte and bit order for all its constituent data types. The ANSI C header files include machine independent macros to read and write fields of those data types correctly regardless of the byte and bit order of the development host machine. This object module format has been proven by active use in the LIFE project at PRPA for over a year, during which time it has become quite stable. Linking of these modules is supported by a generic linker that has also been proven by over two years of active use in the LIFE project.

One might wonder about the tradeoffs involved in providing such a level of generality, such as the cost in linking time of dealing with arbitrary length bitfields at arbitrary positions in bitstrings, and of accessing multi-byte quantities in a manner that is not affected by the byte order of the host machine. It would be reasonable to expect development tools accessing object modules to be slowed down because of this. While this may be so, we at PRPA have never had occasion to measure the slowdown because the steps in the compilation trajectory involving accessing the object module, especially linking, have always proven to take a small percentage of the time required for a program to go through the complete trajectory from source code to executable (for LIFE). In short, accessing the object module is not a significant factor in compilation time. Even cutting link time in half would

not be very noticeable. But to give some perspective, a link of a 1.3 MB program[1] with 44 modules and 1913 unique global symbols[2], took 105 seconds out of a compilation time of 45 minutes on a Sun SPARCstation2 (less than 4%).

While every effort has been made to avoid hard limits on table sizes in the object module format, some limits are imposed by the sizes of the integers chosen to index these tables. These integers have been chosen to be large enough to more than cover any foreseeable need. The limits affecting the object module are summarized in table D-3.

Table D-3Summary of Size Limits

Logical Data Type	Size of Index (bits)	Limit Imposed
SectionId	8	255 sections
StringId	24	16 MB combined length of all strings
Address	64	$2x10^{10}$GB address space
BlockSize	32	2 GB cache block size
FileAddr	32	4 GB object module size
BitAddr	32	4 Gbit bitstring length
FieldWidth	8	255 bit bitfield in bitstring
MemWidth	16	65535 bit memory width
SourceId	16	65535 source files (compilation units)

In conclusion, a very generic and portable object module format has been developed and proven in conjunction with the LIFE project, but is not restricted to LIFE. Its generality allows it to support general purpose architectures as well as architectures that cannot be supported by current object module formats (a.out and COFF). Its portability allows object modules created on one host machine to be used unmodified on any host, a feature even COFF cannot provide. The costs of providing this generality and portability are small and well spent.

D.9 REFERENCES

[1] *"a.out - assembler and link editor output format"*, SunOS UNIX man pages, section 5.

[2] *"coff - common assembler and link editor output"*, SunOS UNIX man pages, section 5.

[3] *"LIFE Technology Assessment Report"*, Philips Research Palo Alto Technical Report #9201.

1. Th program is the SPEC benchmark espresso, and the 1.3 MB is the size of the SPARC executabl produc d by the LIFE compiler for it's profiling run on the SPARCstation (phase 1). The LIFE executable after compilation for LIFE (phase 2), scheduling and assembling, was 12 MB in siz . The .c files comprising the source code have a total of over 13000 lines.
2. The numb r of symbols in the global symbol table after linking.